# Assignment 8: Process Control and Communication

CS3423 - Systems Programming — Rocky Slavin - UTSA

For this assignment, you will use **C**'s various I/O, and IPC functions to create your own rudimentary shell with the following capabilities:

- Execution of commands with arguments
- Input redirection
  (e.g., '`cat < in.txt`')
- Output redirection, including append
  (e.g., '`echo hello > out.txt`' or '`echo world >> out.txt`')
- Pipelining between **two commands**
  (e.g., '`cat out.txt | grep hello`')
- Backgrounding
  (i.e., the shell does not wait for the command to complete if `&` is used at the end of the command)
- Ideally, your program should *all of the above* simultaneously
  (e.g., '`cat < in.txt | grep hello > out.txt &`')
- Your program should continue to prompt for commands until the command '`exit`' is entered, upon which the shell will terminate.

You may assume the following:

- Special characters (`>, >>, <, |, &`) will always be surrounded by a spaces.
- Other shell features not described above do not need to be implemented (e.g., history, file name completion, short circuiting, etc).

## Assignment Data

To reduce much of the work, I am providing a command parser which can be found in `/usr/local/courses/rslavin/cs3423/Fall18/assign8`. The parser, `cmdparse.c`, provides a function, `cmdparse()` which takes a string and produces a `CMD` struct as defined in `cmdparse.h`. You can then use the struct to identify the different components of the compound command as well as what functionality is detected (e.g., redirection, pipelining, etc). Refer to the `cmdparse.c` source code documentation for details. You may set `DEBUG` to `TRUE` in `cmdparse.c` to print useful parsing data for debugging.

Do not copy this code to your assignment file. Instead, link to it when compiling. When graded, `cmdparse.c` and `cmdparse.h` will be copied into the directory before compilation.

Along with the parser, there is a `data` directory containing a few Python scripts useful for testing backgrounding and input redirection.

## Notes on Features and Suggested Order of Implementation

### 1 Single Command

First, you should make your shell be able to execute a command (i.e., child process) with whatever arguments are provided. After the command finishes executing, the shell should prompt the user for the next command.

If a blank command is entered (i.e., the user presses enter immediately), a new prompt should appear and continue as normal.

If the command cannot be parsed, the shell should print '`parse error`' to `STDERR` and continue with a new prompt.

**Hint:** Check `cmdparse.h`

Example command:
`$ ls -l -a`

### 2 Backgrounding

Once your program can execute a command, wait for the command to complete, and prompt for a new command, backgrounding is the next step. Backgrounding is closely related to execution because, in order for a single command to work correctly, the shell must wait for the child process to complete before prompting again. To implement backgrounding, simply stop the shell from waiting for the subprocess to complete and immediately prompt for the next command.

Example command:
`$ python3 data/bgtest.py &`

### 3 Pipelining

Pipelining involves redirecting one file descriptor to another. Use your knowledge of pipes and redirection to allow communication between the two processes.

Remember that, unless backgrounding is used, the shell must not continue until *both* of the two commands have completed execution.

Example command:
`$ cat data/intest.txt | grep sample`

### 4 Redirect STDIN and STDOUT

Redirection of `STDIN` and `STDOUT`, including append (`>>`) should be implemented similarly, but separately. If done correctly, redirection should only be implemented in one place, even with pipelining taken into account.

If a new file is created with output redirection, the permissions `0644` should be used.

If a redirection file is not found, print "`unable to open file for redirection`" to `STDERR` and prompt for the next command.

Both input and output redirection should work at the same time for a single command, including with pipelining.

Example commands:

```
$ echo hello > outfile.txt
$ cat < outfile.txt
$ cat < outfile.txt > outfile.copy
$ cat < outfile.copy | grep hello > out.txt
```

## Compiling Your Program

Your submission will include a makefile so the following command can be used to compile your code.

```
$ make assign8
```

This should produce an executable file named **myshell**.

## Program Files

Your submission should consist of up to three files:

- `assign8.c` - the main file which is compiled **(required)**
- `assign8.h` - an optional header file if necessary
- `makefile` - the makefile to make the `myshell` executable **(required)**

## Verifying Your Program

Your shell should run on the fox machines by invoking the following command.

```
$ myshell
```

This should produce a prompt, '`$ `' (notice the space at the end), and take commands from the user. After a command is successfully executed, a new prompt should appear (unless backgrounding is detected, in which case the prompt should appear and take commands immediately without waiting for the command to complete).

It is up to you to test the functionality of your shell based on the requirements at the beginning of this document.

## Submission

Turn your assignment in via Blackboard. Your zip file, named **`LastnameFirstname.zip`** should contain only your **`makefile`**, **`assign8.c`**, and possibly **`assign8.h`**. Do not include **`cmdparse.c`** nor **`cmdparse.h`**